CS 33

Data Representation (Part 3)

Byte Ordering

- **Four-byte integer**
  - 0x76543210
- **Stored at location 0x100**
  - which byte is at 0x100?
  - which byte is at 0x103?

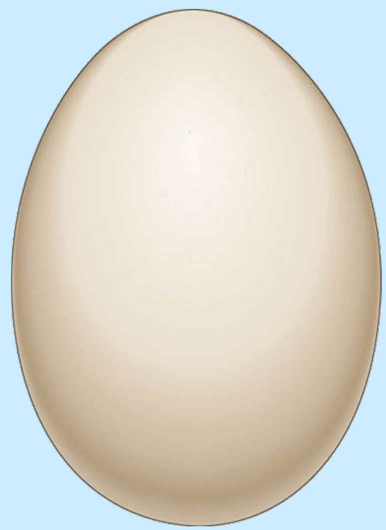
?

10	32	54	76
0x100	0x101	0x102	0x103

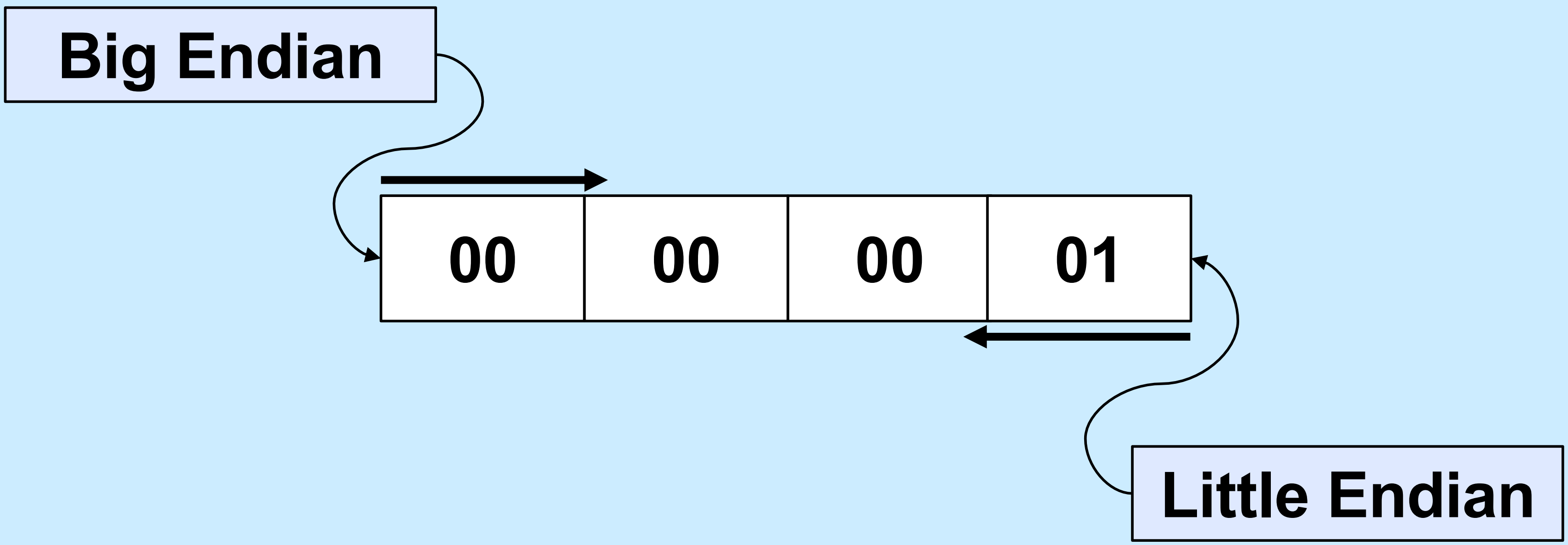
Little-endian

76	54	32	10
0x100	0x101	0x102	0x103

Big-endian



Byte Ordering (2)



Quiz 1

```
int main() {
   long x=1;
   func((int *)&x);
   return 0;
}

void func(int *arg) {
   printf("%d\n", *arg);
}
```

What value is printed on a big-endian 64-bit computer?

a) 1
b) 0
c) 2^{32}
d) 2^{32}-1

Which Byte Ordering Do We Use?

```
int main() {
    unsigned int x = 0x03020100;
    unsigned char *xarray = (unsigned char *)&x;
    for (int i=0; i<4; i++) {
        printf("%02x", xarray[i]);
    }
    printf("\n");
    return 0;
}
```

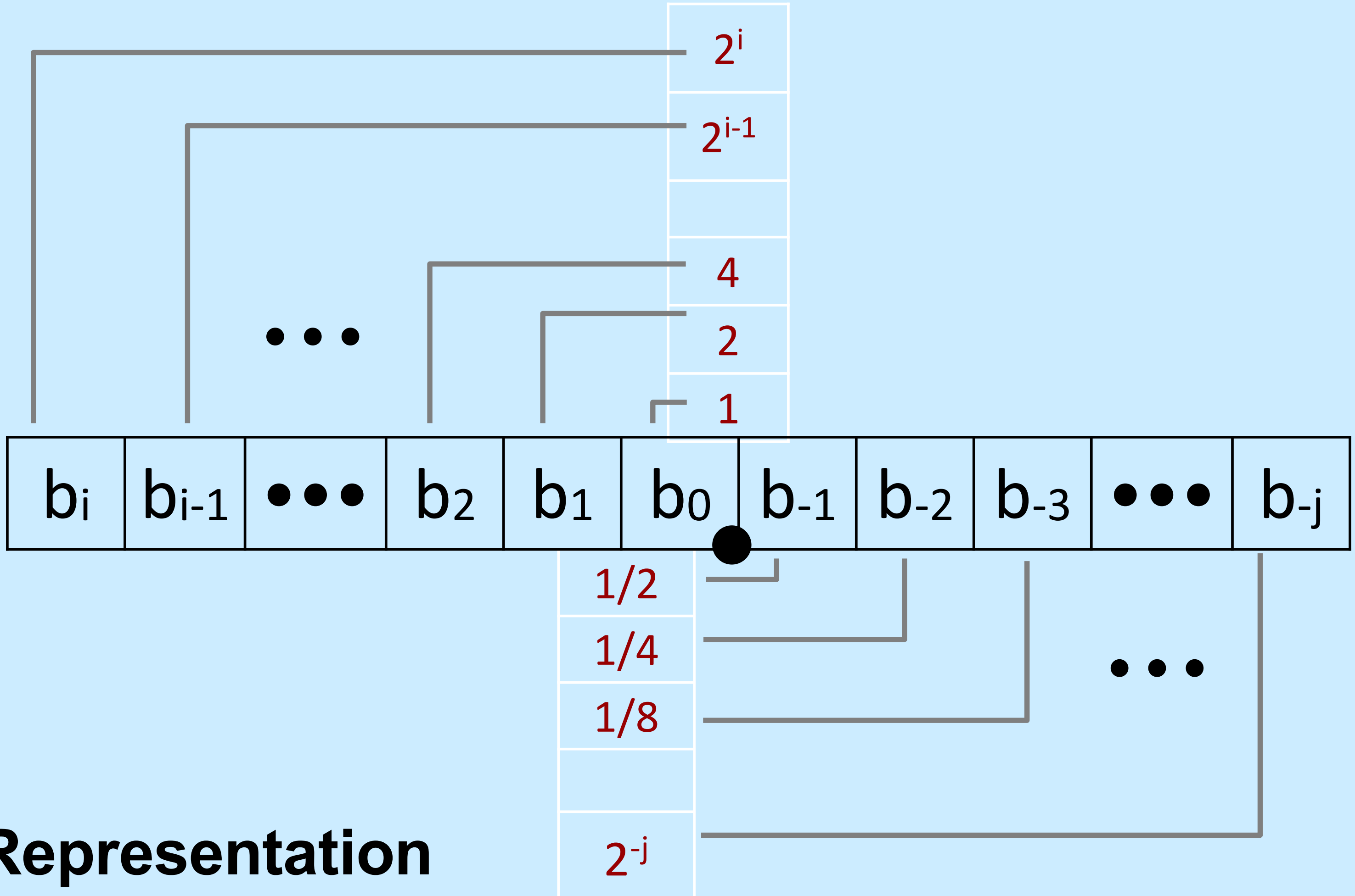
Possible results:

```
00010203
03020100
```

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- **Representation**
  - **bits to right of “binary point” represent fractional powers of 2**
  - **represents rational number:** $\sum_{k=-j}^{i} b_k \times 2^k$

Representable Numbers

- **Limitation #1**
  - can exactly represent only numbers of the form $n/2^k$
    - » other rational numbers have repeating bit representations
  - value — representation
    - » 1/3 — $0.0101010101[01]\ldots_2$
    - » 1/5 — $0.001100110011[0011]\ldots_2$
    - » 1/10 — $0.0001100110011[0011]\ldots_2$
- **Limitation #2**
  - just one setting of decimal point within the *w* bits
    - » limited range of numbers (very small values? very large?)

IEEE Floating Point

- IEEE Standard 754
  - established in 1985 as uniform standard for floating point arithmetic
    - » before that, many idiosyncratic formats
  - supported on all major CPUs
- Driven by numerical concerns
  - nice standards for rounding, overflow, underflow
  - hard to make fast in hardware
    - » numerical analysts predominated over hardware designers in defining standard

Floating-Point Representation

- **Numerical Form:**

$$(-1)^s \; M \;\; 2^E$$

  - **sign bit s determines whether number is negative or positive**
  - **significand M normally a fractional value in range [1.0,2.0)**
  - **exponent E weights value by power of two**
- **Encoding**
  - **MSB `s` is sign bit s**
  - **`exp` field encodes E (but is not equal to E)**
  - **`frac` field encodes M (but is not equal to M)**

s	exp	frac

Precision options

- **Single precision: 32 bits**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**

s	exp	frac
1	11-bits	52-bits

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15-bits	64-bits

“Normalized” Values

- **When: exp ≠ 000…0 and exp ≠ 111…1**

- **Exponent coded as biased value: E = Exp – Bias**
  - **exp: unsigned value `exp`**
  - **bias = 2^{k-1} - 1, where k is number of exponent bits**
    - » **single precision: 127 (Exp: 1…254, E: -126…127)**
    - » **double precision: 1023 (Exp: 1…2046, E: -1022…1023)**

- **Significand coded with implied leading 1: M = $1.xxx…x_2$**
  - **`xxx…x`: bits of `frac`**
  - **minimum when `frac=000…0` (M = 1.0)**
  - **maximum when `frac=111…1` (M = 2.0 – ε)**
  - **get extra leading bit for “free”**

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  M = `1.`$\underline{\texttt{1101101101101}}_2$

  frac = $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- **Exponent**

  E = **13**

  bias = **127**

  exp = **140** = 10001100_2

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

- Condition: exp = `000…0`
- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0:
  $M = 0.xxx\ldots x_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - » represents zero value
    - » note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - » numbers closest to 0.0
    - » equispaced

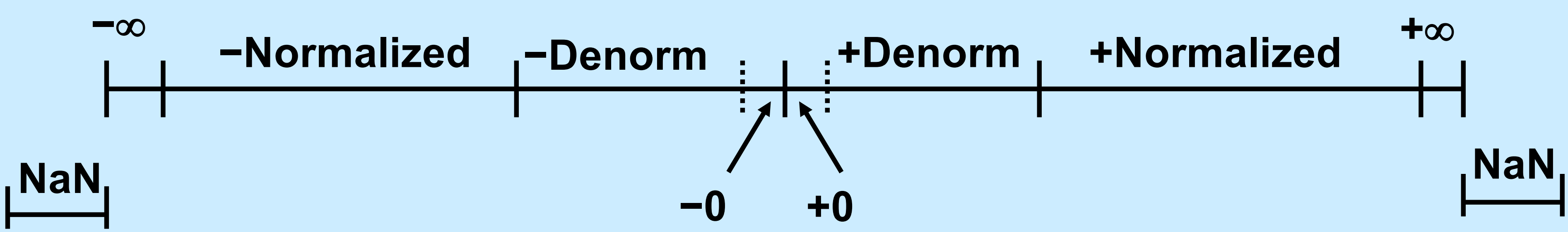
Special Values

- **Condition: `exp` = 111…1**

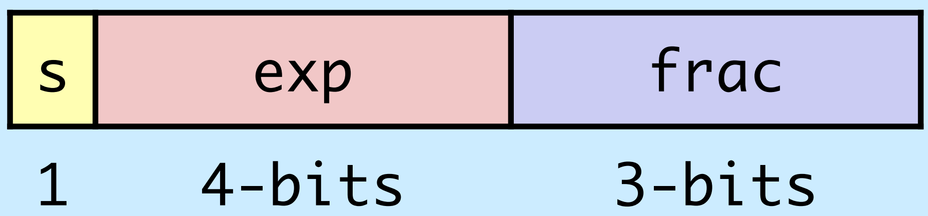
- **Case: `exp` = 111…1, `frac` = 000…0**
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - e.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- **Case: `exp` = 111…1, `frac` ≠ 000…0**
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating-Point Encodings



Tiny Floating-Point Example



- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`

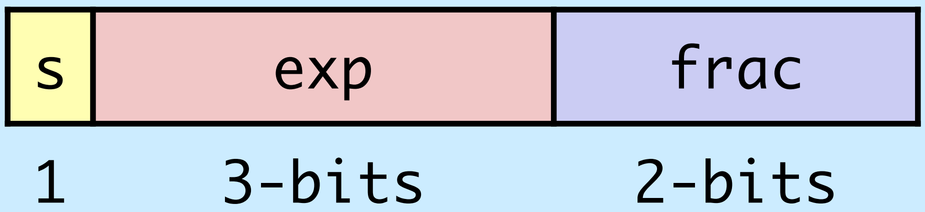
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

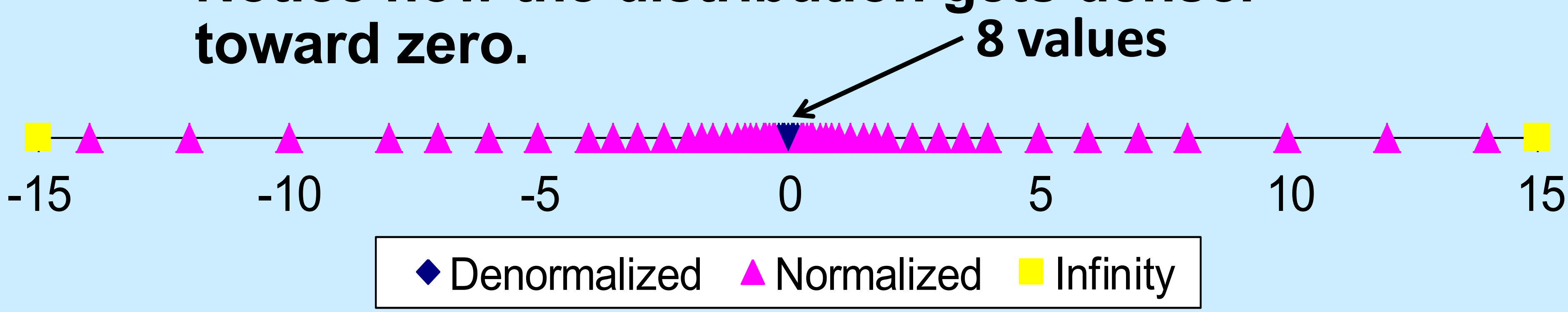
	s exp frac	E	Value	
	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	largest norm
	0 1111 000	n/a	inf	

Distribution of Values

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is 2^{3-1}-1 = 3**

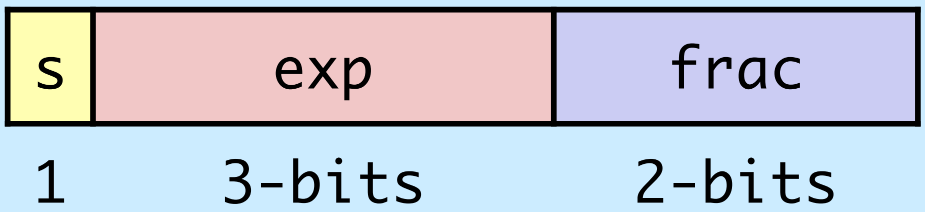


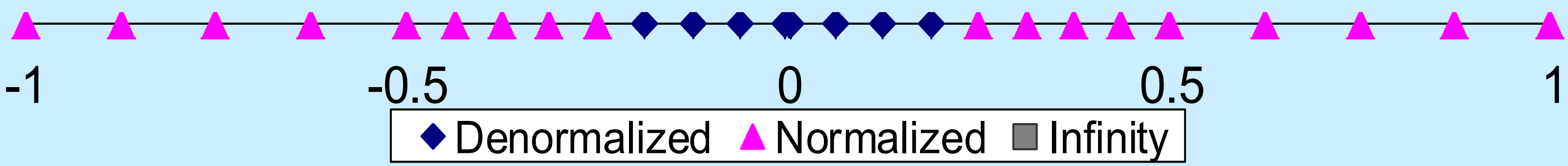
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

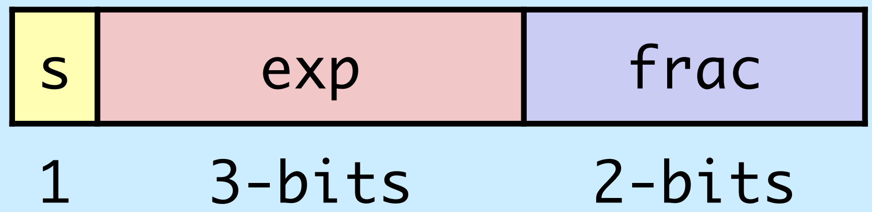
- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3





Quiz 2

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3



What number is represented by 0 010 10?

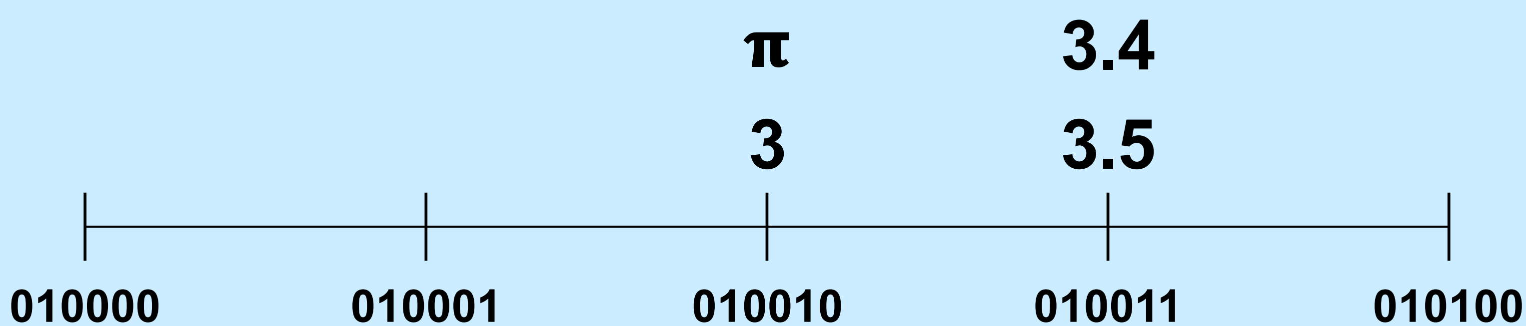
a) 3
b) 1.5
c) .75
d) none of the above

Mapping Real Numbers to Float

- The real number 3 is represented as
  0 100 10
- The real number 3.5 is represented as
  0 100 11
- How is the real number 3.4 represented?

  0 100 11
- How is the real number π represented?

  0 100 10

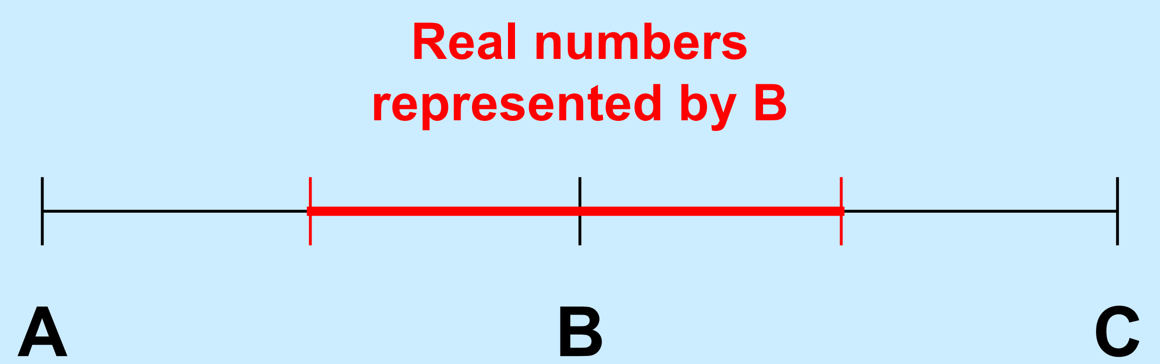


Mapping Real Numbers to Float

- If R is a real number, it's mapped to the floating-point number whose value is closest to R
- What if it's midway between two values?
  - rounding rules determine outcome

Floats are Sets of Values

- If A, B, and C are successive floating-point values
  - e.g., 010001, 010010, and 010011
- B represents all real numbers from midway between A and B through midway between B and C



Significance

- **Normalized numbers**
  - for a particular exponent value E and an S-bit significand, the range from 2^E up to 2^{E+1} is divided into 2^S equi-spaced floating-point values
    - » thus each floating-point value represents $1/2^S$ of the range of values with that exponent
    - » all bits of the significand are important
    - » we say that there are S significant bits – for reasonably large S, each floating-point value covers a rather small part of the range
      - high accuracy
      - for S=23 (32-bit float), accurate to one in 2^{23} (.0000119% accuracy)

Significance

- **Unnormalized numbers**
  - high-order zero bits of the significand aren't important
  - in 8-bit floating point, 0 0000 001 represents 2^{-9}
    - » it is the only value with that exponent: 1 significant bit (either 2^{-9} or 0)
    - » 50% accuracy
  - 0 0000 010 represents 2^{-8}
    0 0000 011 represents $1.5*2^{-8}$
    - » only two values with exponent -8: 2 significant bits (encoding those two values, as well as 2^{-9} and 0)
    - » 25% accuracy
  - fewer significant bits means less accuracy
  - 0 0000 001 represents a range of values from $.5*2^{-9}$ to $1.5*2^{-9}$

+/− Zero

- Only one zero for ints
  - an int is a single number, not a range of numbers, thus there can be only zero
- Floating-point zero
  - a range of numbers around the real 0
  - it really matters which side of 0 we're on!
    - » a very large negative number divided by a very small negative number should be positive

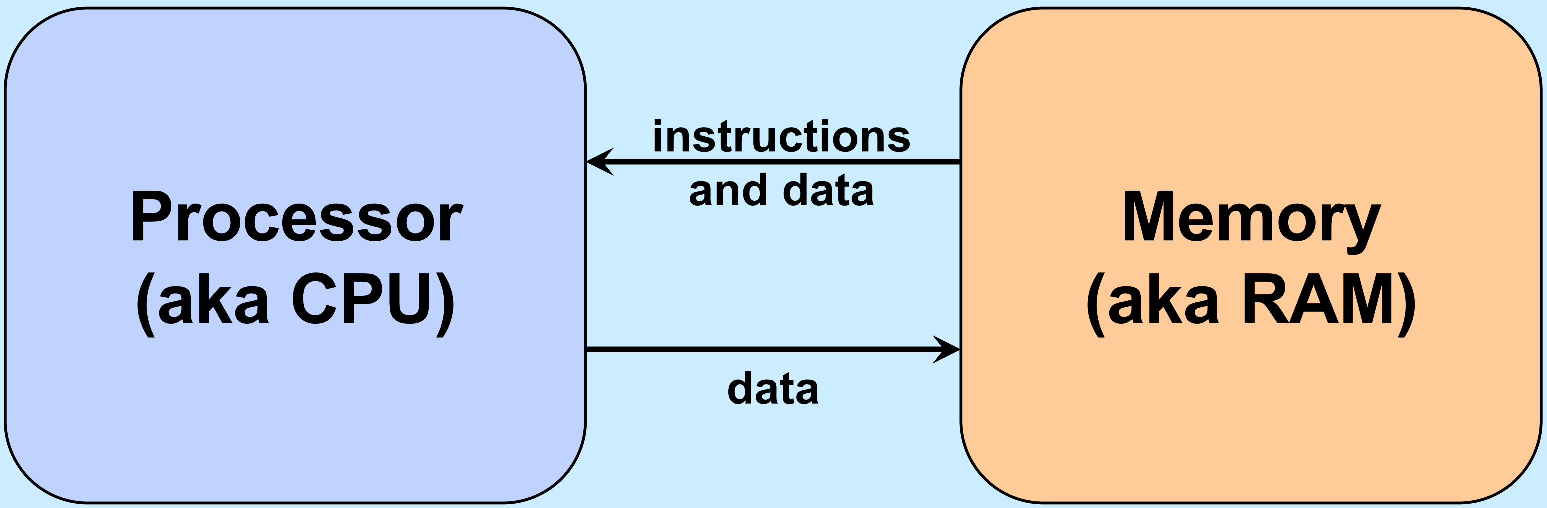
      $-\infty/-0 = +\infty$
    - » a very large positive number divided by a very small negative number should be negative

      $+\infty\ /-0 = -\infty$

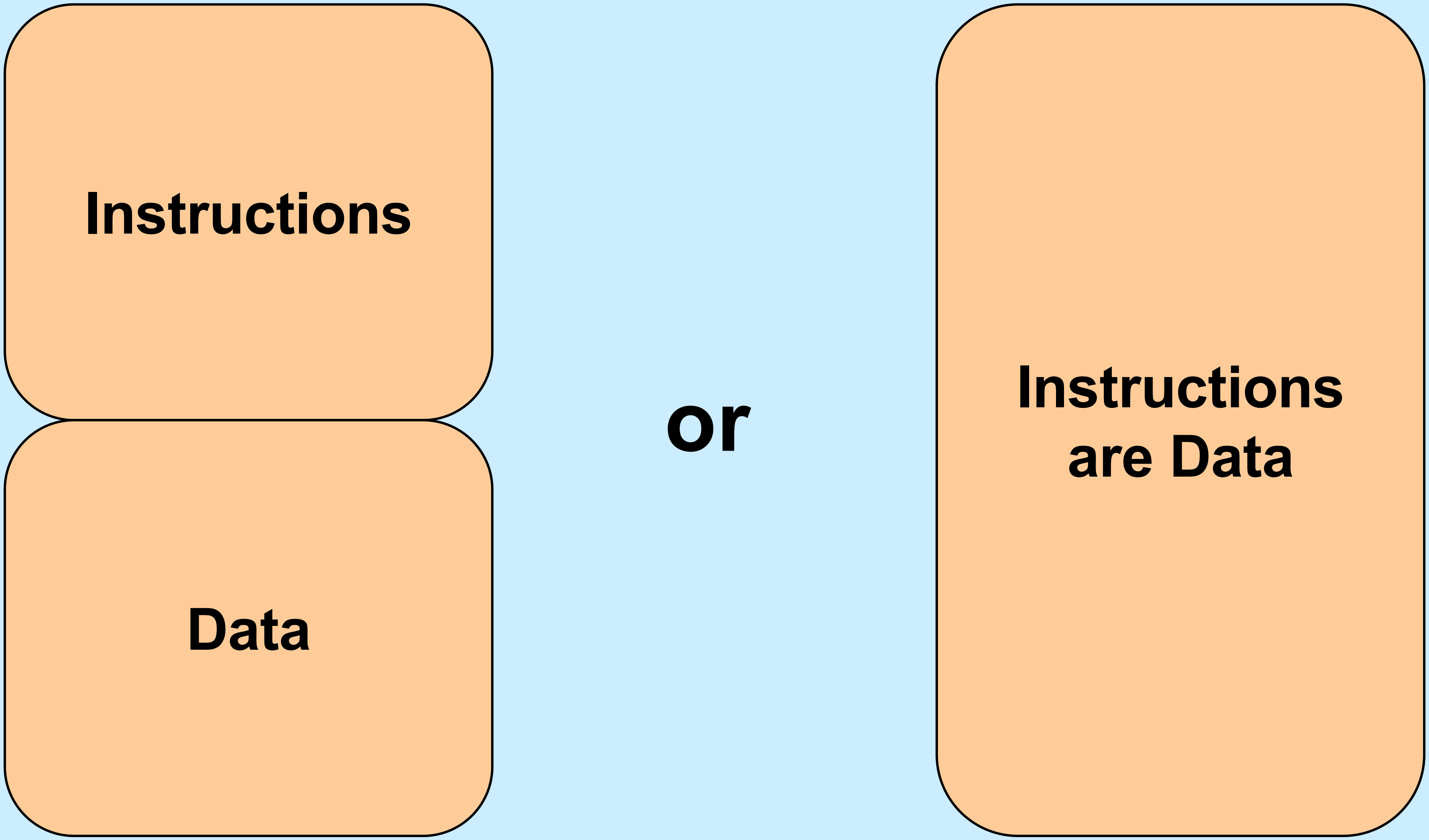
CS 33

Intro to Machine Programming

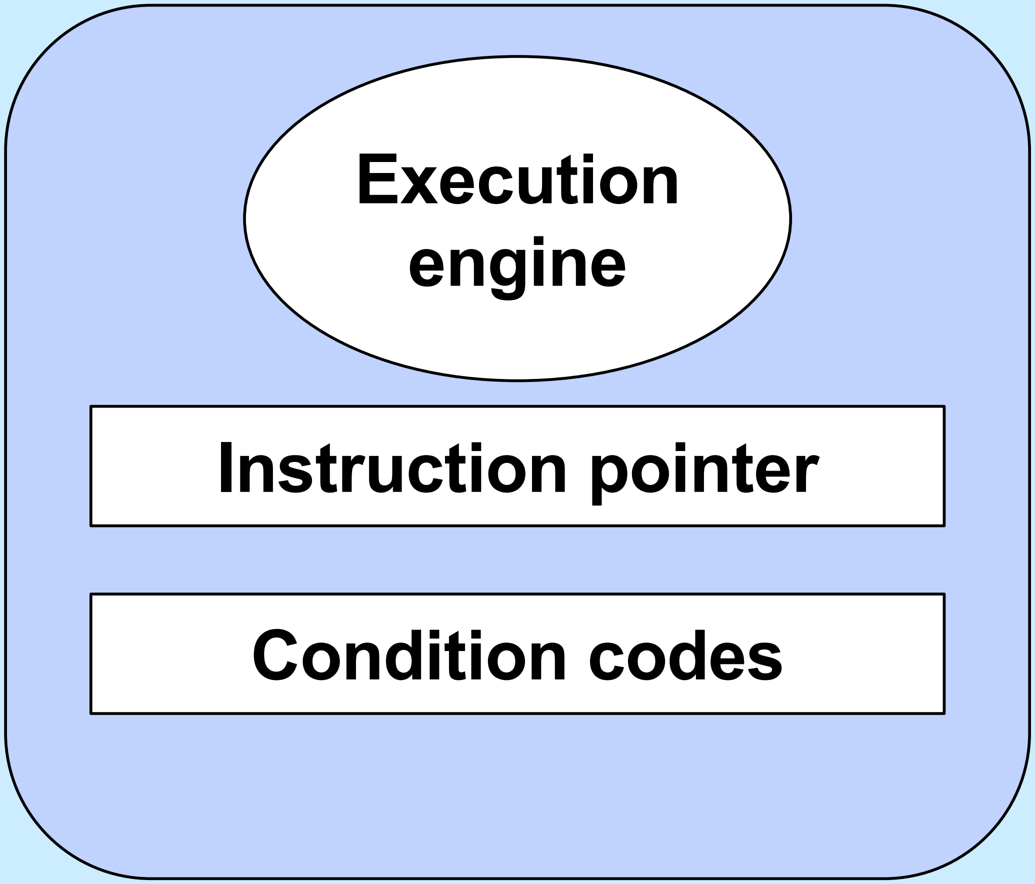
Machine Model



Memory



Processor: Some Details



Processor: Basic Operation

```
while (forever) {
  fetch instruction IP points at
  decode instruction
  fetch operands
  execute
  store results
  update IP and condition code
}
```

Instructions ...

Op code	Operand1	Operand2	...

Operands

- Form
  - immediate vs. reference
    - » value vs. address
- How many?
  - 3
    - » add a,b,c
      - c = a + b
  - 2
    - » add a,b
      - b += a

Operands (continued)

- **Accumulator**
  - special memory in the processor
    - known as a *register*
    - fast access
  - allows single-operand instructions
    - add a
      - acc += a
    - add b
      - acc += b

From C to Assembler ...

```
a = (b + c) * d;

    mov    b,%acc
    add    c,%acc
    mul    d,%acc
    mov    %acc,a
```

```
if (a<b)
    c = 1;
else
    d = 1;

    cmp    a,b
    jge    .L1
    mov    $1,c
    jmp    .L2
.L1
    mov    $1,d
.L2
```

immediate operand

immediate operand

Condition Codes

- Set of flags giving status of most recent operation:
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- Set implicitly by arithmetic instructions
- Set explicitly by compare instruction
  - cmp a,b
    - » sets flags based on result of b-a

Examples (1)

- Assume 32-bit arithmetic

- x is `0x80000000`
  - TMIN if interpreted as two's-complement
  - 2^{31} if interpreted as unsigned
- x-1 (`0x7fffffff`)
  - TMAX if interpreted as two's-complement
  - 2^{31}-1 if interpreted as unsigned
  - zero flag is not set
  - sign flag is not set
  - overflow flag is set
  - carry flag is not set

Examples (2)

- **x is `0xffffffff`**
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- **x+1 (`0x00000000`)**
  - zero under either interpretation
  - zero flag is set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Examples (3)

- x is `0xffffffff`
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- x+2 (`0x00000001`)
  - (+)1 under either interpretation
  - zero flag is not set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Quiz 3

- **Set of flags giving status of most recent operation:**
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- **Set explicitly by compare instruction**
  - cmp a,b
    - » sets flags based on result of b-a

Which flags are set to one by "cmp 2,1"?

a) overflow flag only
b) carry flag only
c) sign and carry flags only
d) sign and overflow flags only
e) sign, overflow, and carry flags

Jump Instructions

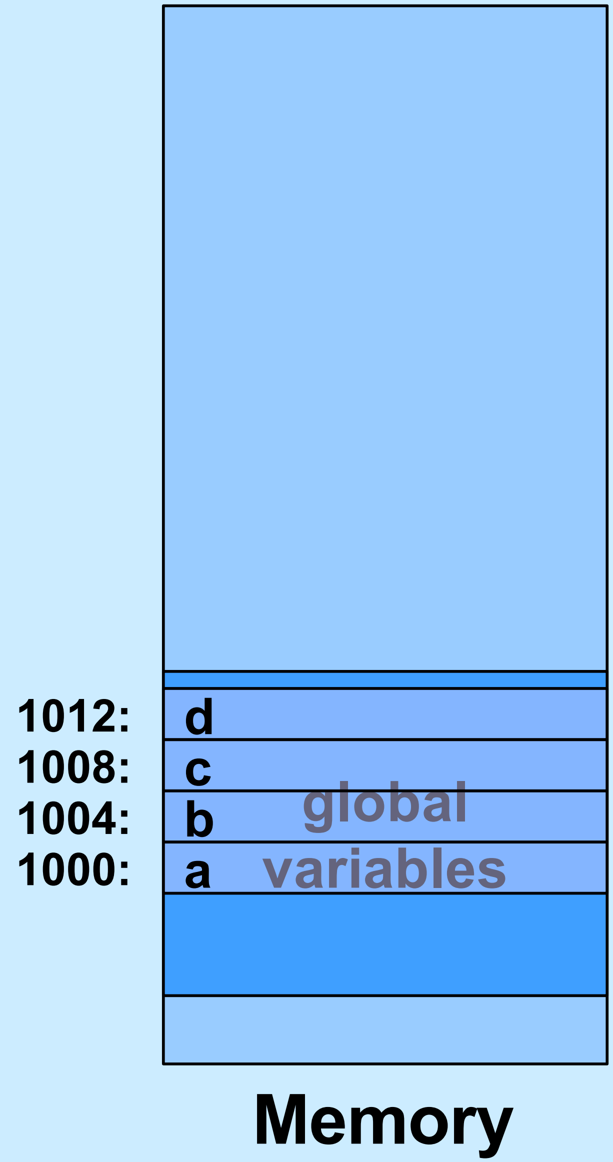
- **Unconditional jump**
  - just do it
- **Conditional jump**
  - to jump or not to jump determined by condition-code flags
  - field in the op code indicates how this is computed
  - in assembler language, simply say
    - » je
      - jump on equal
    - » jne
      - jump on not equal
    - » jg
      - jump on greater than (signed)
    - » etc.

Addresses

```
int a, b, c, d;

int main() {
    a = (b + c) * d;
    ...
}
```

```
mov     b,%acc
add     c,%acc
mul     d,%acc
mov     %acc,a
```

```
mov     1004,%acc
add     1008,%acc
mul     1012,%acc
mov     %acc,1000
```



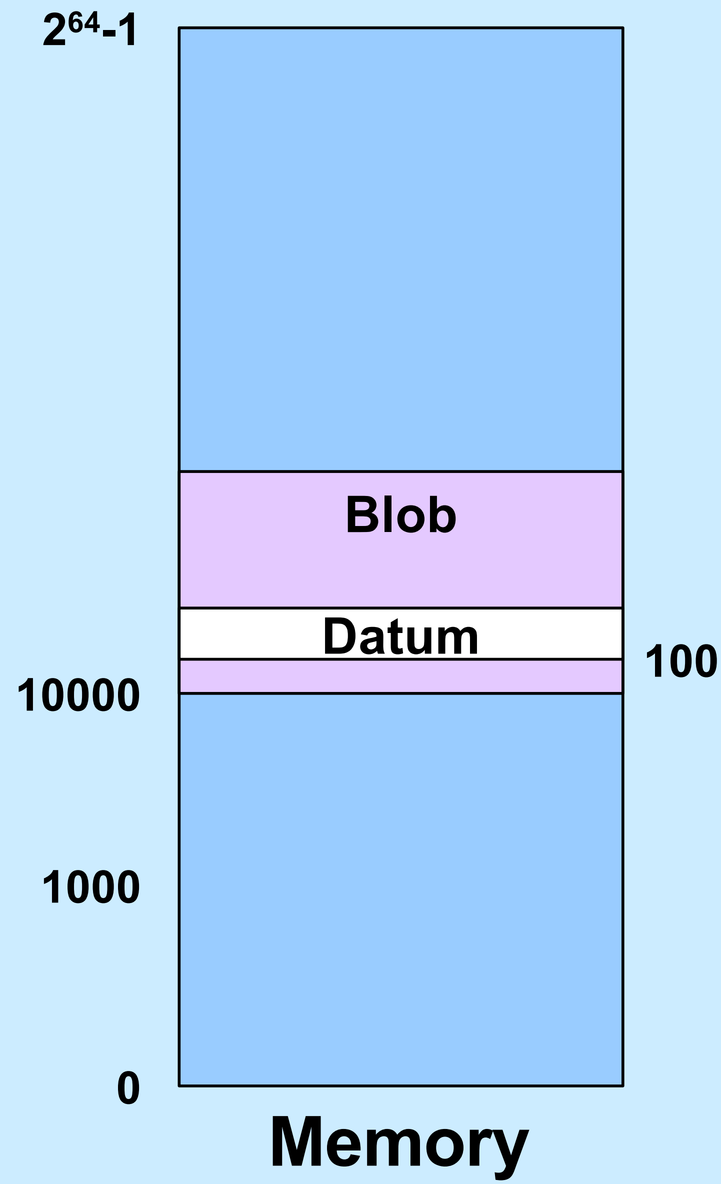
Addresses

```
int b;

int func(int c, int d) {
    int a;
    a = (b + c) * d;
    ...
}


    mov    ?,%acc
    add    ?,%acc
    mul    ?,%acc
    mov    %acc,?
```

- One copy of *b* for duration of program's execution
  - *b*'s address is the same for each call to *func*
- Different copies of *a*, *c*, and *d* for each call to *func*
  - addresses are different in each call

Relative Addresses

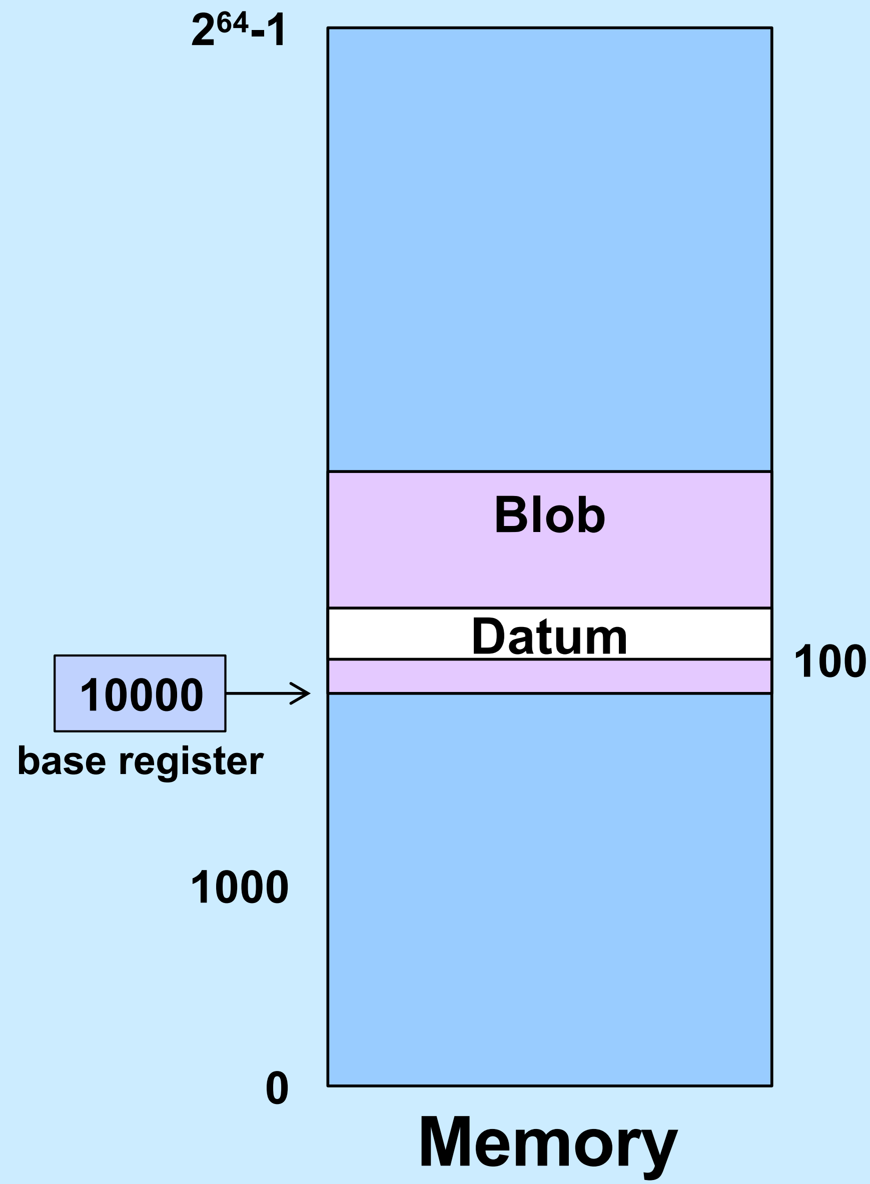
- **Absolute address**
  - actual location in memory
- **Relative address**
  - offset from some other location



- Blob's absolute address is 10000
- Datum's relative address (to Blob) is 100
  - its absolute address is 10100

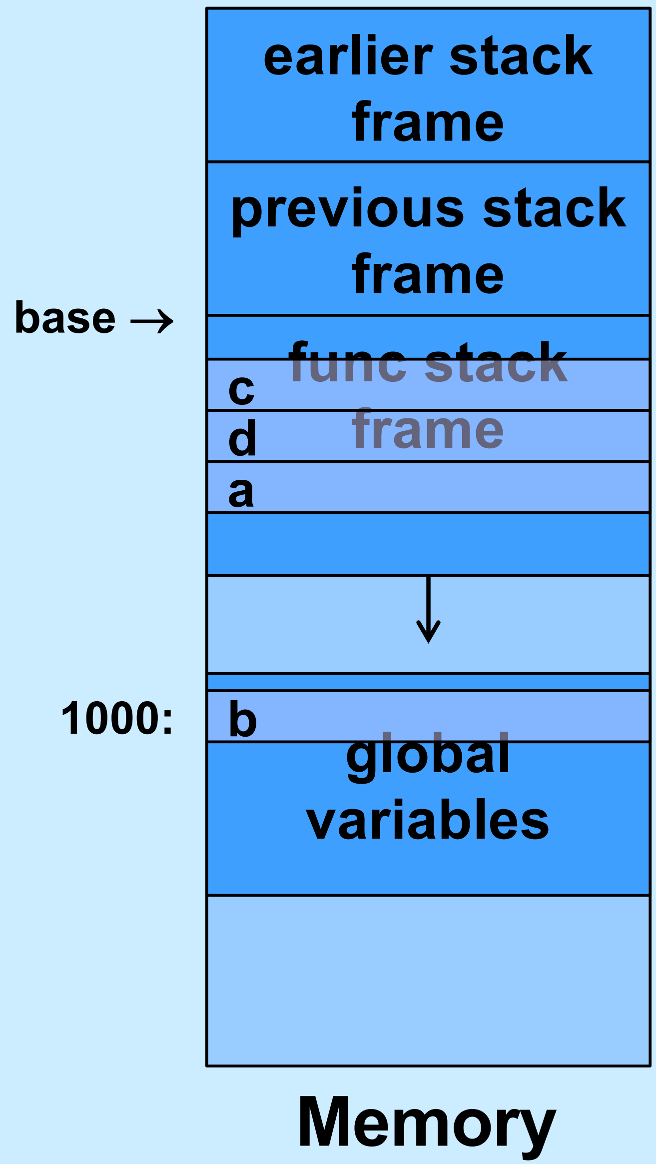
Base Registers

```
mov $10000, %base
mov $10, 100(%base)
```



Addresses

```
long b;

int func(long c, long d) {
    long a;
    a = (b + c) * d;
    ...
}

    mov    1000,%acc
    add    -8(%base),%acc
    mul    -16(%base),%acc
    mov    %acc,-24(%base)
```

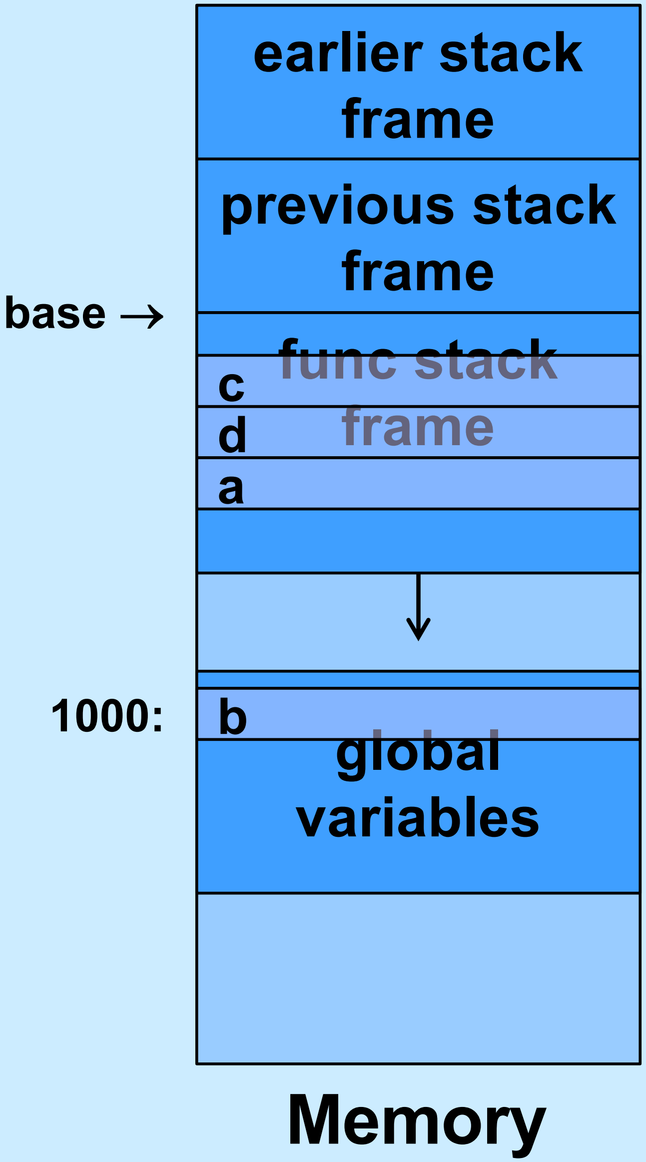


Quiz 4

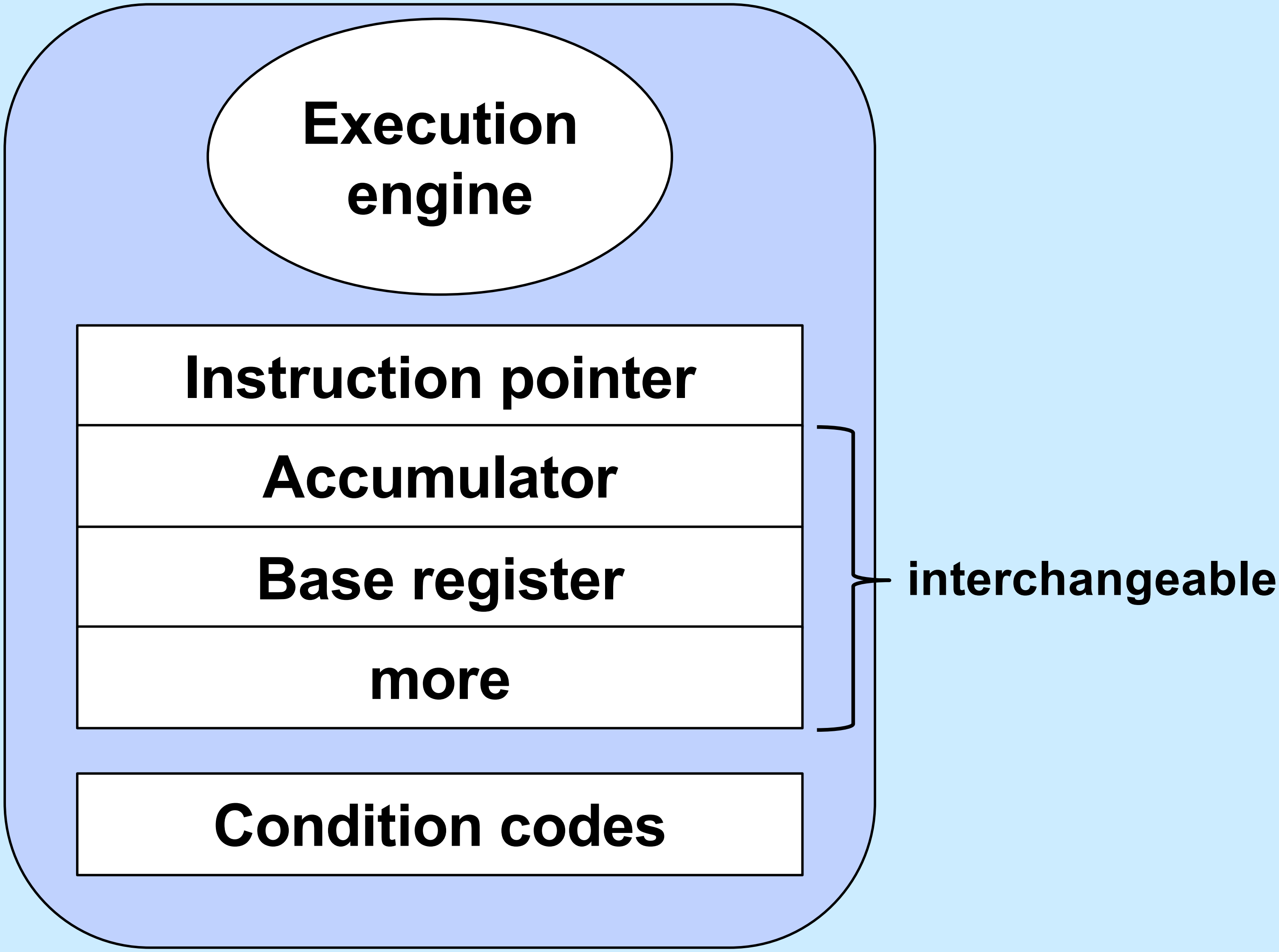
Suppose the value in *base* is 10,000. What is the address of *c*?

a) 10,016
b) 10,008
c) 9992
d) 9984

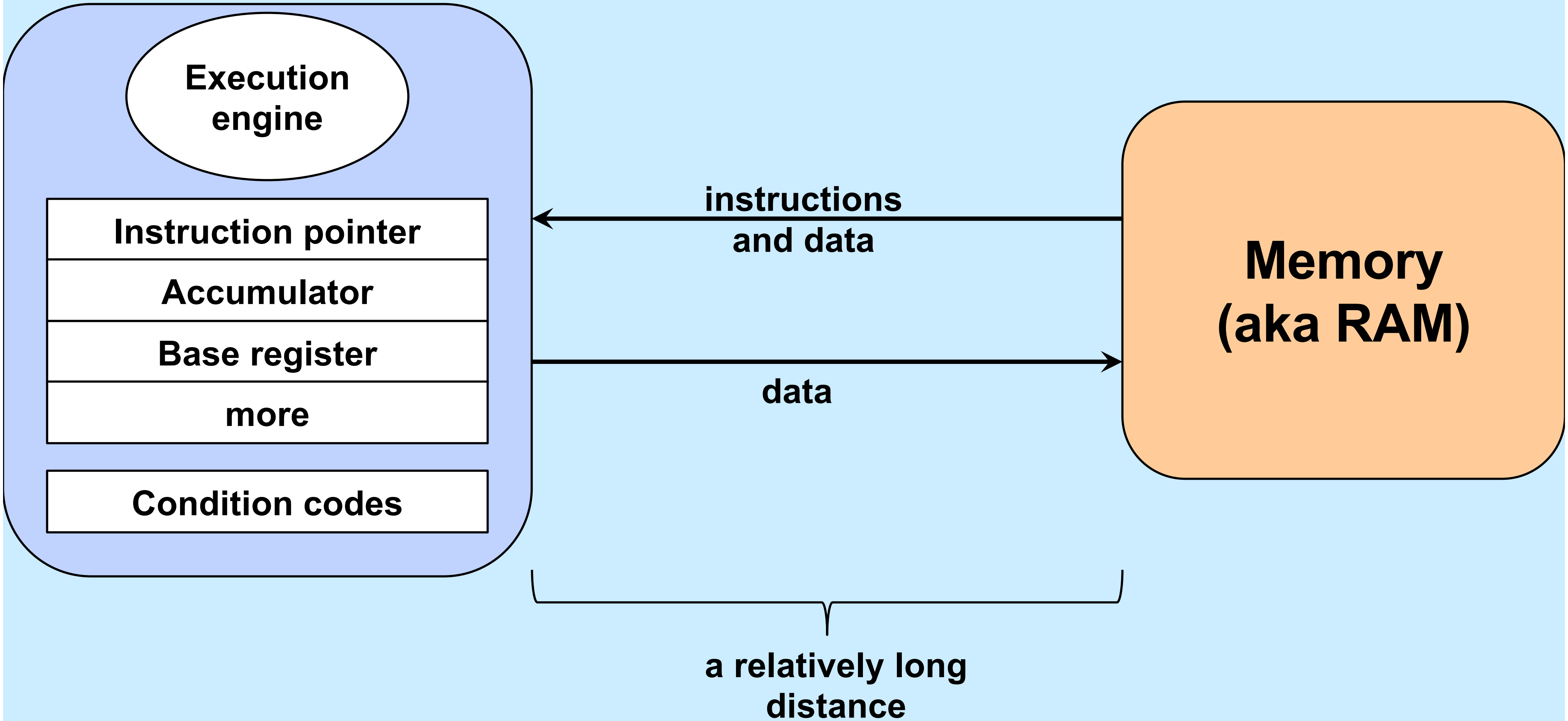
```
mov     1000,%acc
add     -8(%base),%acc
mul     -12(%base),%acc
mov     %acc,-16(%base)
```



Registers


Execution engine
Instruction pointer
Accumulator
Base register
more
Condition codes
interchangeable

Registers vs. Memory



Intel x86

- Intel created the 8008 (in 1972)
- 8008 begat 8080
- 8080 begat 8086
- 8086 begat 8088
- 8086 begat 286
- 286 begat 386
- 386 begat 486
- 486 begat Pentium
- Pentium begat Pentium Pro
- Pentium Pro begat Pentium II
- ad infinitum

IA32

2^{64}

- 2^{32} used to be considered a large number
  - one couldn't afford 2^{32} bytes of memory, so no problem with that as an upper bound
- Intel (and others) saw need for machines with 64-bit addresses
  - devised IA64 architecture with HP
    - » became known as Itanium
    - » very different from x86
- AMD also saw such a need
  - developed 64-bit extension to x86, called x86-64
- Itanium flopped
- x86-64 dominated
- Intel, reluctantly, adopted x86-64

Why Intel?

- Most CS Department machines are Intel
- An increasing number of personal machines are not
  - Apple has switched to ARM
  - packaged into their M1, M2, etc. chips
    - » "Apple Silicon"
- Intel x86-64 is very different from ARM64 — internally
- Programming concepts are similar
- We cover Intel; most of the concepts apply to ARM